Stable Value

Rémi Forax

Oracle Developer Week - May 2025

https://github.com/forax/stable-value

Two questions:

Lazy initialization ?

Lazy constant initialization ?

Part I

Lazy initialization in Java ?

Why lazy init ??

In any languages

- Defer the cost until the value is needed
  - Avoid not useful computation
  - Avoid not useful storage

Strawman code (bad !)

Initialize with null + null check when the value is needed

```
public class MyClass {
  private Database db;    // not final

  public Database getDatabase() {
    if (db != null) {
      return db;
    }
    return db = new Database(…);
  }
}
```

Access with : myClass.getDatabase()

Thread safety !

The code is not thread-safe

- Can create multiple Database objects !
- Can create a Database object not fully initialized !

```
public class MyClass {
  private Database db;

  public Database getDatabase() {
    if (db != null) {
      return db;
    }
    return db = new Database(…);
  }
}
```

Thread safe publication

In pseudo assembler

```
public class MyClass {
  private Database db;

  public Database getDatabase() {
    if (db != null) {
      return db;
    }
    var tmp = new Database   // pseudo-assembler
    tmp.field1 = …     // write in memory
    tmp.field2 = …     // write in memory
    …
    db = tmp;          // write in memory
    return tmp;
  }
}
```

Can those writes be re-organized ?

TSO vs Weak model

On an Intel/AMD 64bits, stores are retired in the assembly order

- Total Store Order, stores can not be reordered

On a ARM aarch64, stores can be reordered

Java Memory Model

Allow Stores re-ordering

- Opt-in memory fences
  - Synchronized block
  - Final field
  - Volatile field

Even on a TSO CPU, JITs can reorder the stores

With a synchronized block

Make sure than the Database instance is fully initialized before being available to another thread

```
private Database db;
private Object lock = new Object();

public Database getDatabase() {
  synchronized(lock) {
    if (db != null) {
      return db;
    }
    var tmp = new db   // pseudo-assembler
    …
    db = tmp;
    return tmp;
  }
}
```

Warning, Warning

All perf measurements have been done on my laptop ! (MacBook Air M2)

Using JMH: https://github.com/openjdk/jmh

Perf – lazy init synchronized

Synchronized is a perf killer :(

```
Benchmark                           Mode  Cnt  Score    Error  Units
LazyInitBench.lazy_synchronized_string  avgt    5  5,413 ± 0,168  ns/op
LazyInitBench.string                avgt    5  0,313 ± 0,001  ns/op
```

Double Check Locking

A design pattern from C++

Try to avoid synchronized cost by adding a null check upfront

Double Check Locking (bad !)

https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

This code has a serious bug !

```
private Database db;
private Object lock = new Object();

public static Database getDatabase() {
  if (db != null) {
    return db;
  }
  synchronized(lock) {
    if (db != null) {
      return db;
    }
    var tmp = new db   // pseudo-assembler
    tmp.field1 = …     // write in memory
    …
    db = tmp;          // write in memory
    return tmp;
  }
}
```

If those writes are re-organized
the object can be published without the
fields initialized

Double Check Locking (good !)

Can avoid the re-organization with volatile !

```
private volatile Database db;
private Object lock = new Object();

public Database getDatabase() {
  if (db != null) {
    return db;             // volatile read
  }
  synchronized(LOCK) {
    if (db != null) {
      return db;
    }
    var tmp = new db       // pseudo-assembler
    tmp.field1 = …         // write
    …
    db = tmp;              // volatile write
    return tmp;
  }
}
```

If Database is immutable (all fields are final) then volatile is not necessary

Perf – lazy init DCL

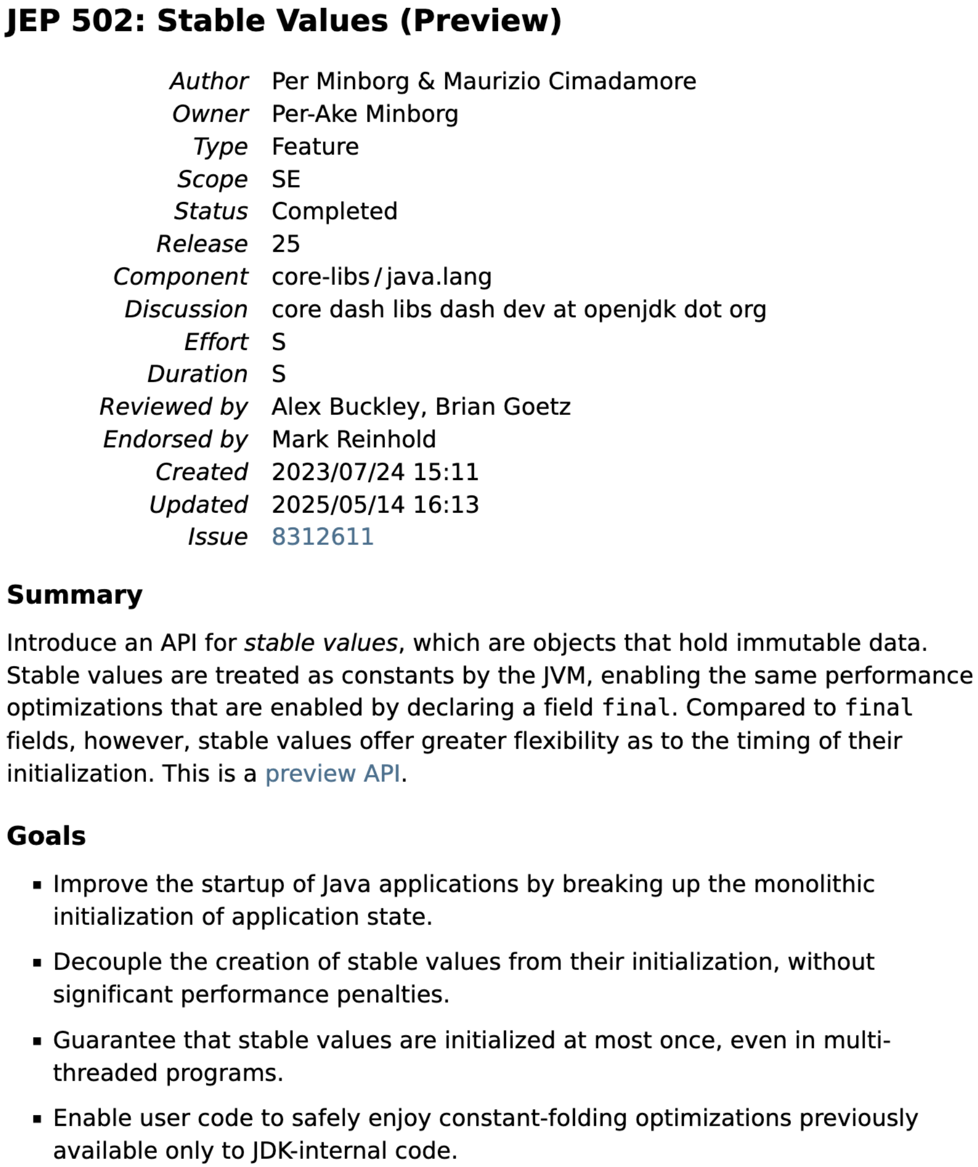
Lazy String init with a DCL is not a constant

```
Benchmark                             Mode  Cnt  Score   Error  Units
LazyInitBench.lazy_dcl_string         avgt    5  0,731 ± 0,009  ns/op
LazyInitBench.lazy_synchronized_string avgt   5  5,413 ± 0,168  ns/op
LazyInitBench.string                  avgt    5  0,313 ± 0,001  ns/op
```

Scala lazy val uses the DCL

Kotlin lazy(SYNCHRONIZED, …) uses the DCL

Stable Value



JEP 502: Stable Values (Preview)

Author	Per Minborg & Maurizio Cimadamore
Owner	Per-Ake Minborg
Type	Feature
Scope	SE
Status	Completed
Release	25
Component	core-libs / java.lang
Discussion	core dash libs dash dev at openjdk dot org
Effort	S
Duration	S
Reviewed by	Alex Buckley, Brian Goetz
Endorsed by	Mark Reinhold
Created	2023/07/24 15:11
Updated	2025/05/14 16:13
Issue	8312611

Summary

Introduce an API for *stable values*, which are objects that hold immutable data. Stable values are treated as constants by the JVM, enabling the same performance optimizations that are enabled by declaring a field `final`. Compared to `final` fields, however, stable values offer greater flexibility as to the timing of their initialization. This is a preview API.

Goals

- Improve the startup of Java applications by breaking up the monolithic initialization of application state.
- Decouple the creation of stable values from their initialization, without significant performance penalties.
- Guarantee that stable values are initialized at most once, even in multi-threaded programs.
- Enable user code to safely enjoy constant-folding optimizations previously available only to JDK-internal code.

java.lang.StableValue

Preview API in Java 25

Provide a simple API to do lazy initialization

- Should be as fast as DCL ?
- Should be as fast as a final field ?

DEMO !
(1)

Stable Value Supplier

High level API

```
private final Supplier<Database> supplier =
    StableValue.supplier(() - >  new Database(… ) );

public Database getDatabase() {
  return supplier.get();
}
```

```
Supplier<T> StableValue.supplier(Supplier<T>)
```

StableValue + .orElse()

Lower level API

```
private final StableValue<Database> value =
    StableValue.of();

public Database getDatabase() {
  return value.orElseGet(() - > new Database(…));
}
```

StableValue.of() + .orElseGet(Supplier<T>)

Perf – stable value

Stable supplier / Stable value

```
Benchmark                               Mode  Cnt  Score   Error  Units
LazyInitBench.lazy_dcl_string           avgt    5  0,731 ± 0,009  ns/op
LazyInitBench.lazy_synchronized_string  avgt    5  5,413 ± 0,168  ns/op
LazyInitBench.stable_supplier_string    avgt    5  0,894 ± 0,003  ns/op
LazyInitBench.stable_value_string       avgt    5  0,829 ± 0,003  ns/op
LazyInitBench.string                    avgt    5  0,313 ± 0,001  ns/op
```

StableValue now

Easier to use than the DCL

Faster than using a synchronized block

but Perf are far from the perf of a final field

- We have work to do :)

Part II

Lazy constant initialization in Java ?

Literal values ?

Literal values are constant

```
void main() {
  System.out.println(42);
  System.out.println("Am i a constant ?");
}
```

```
 0: getstatic     #7    // Field System.out:LPrintStream;
 3: bipush        42
 5: invokevirtual #13   // Method PrintStream.println:(I)V
 8: getstatic     #7    // Field System.out:LPrintStream;
11: ldc           #19   // String Am i a constant ?
13: invokevirtual #21   // Method PrintStream.println:(LString;)V
```

Bytecode produces by javac (using javap)

Static final ?

Static final directly initialized are constant

```
static final int MAGIC = 40 + 2;
static final String STRING = "Am i a constant ?";

void main() {
  System.out.println(MAGIC);
  System.out.println(STRING);
}
```

```
 0: getstatic     #7    // Field System.out:LPrintStream;
 3: bipush        42
 5: invokevirtual #13   // Method PrintStream.println:(I)V
 8: getstatic     #7    // Field System.out:LPrintStream;
11: ldc           #19   // String Am i a constant ?
13: invokevirtual #21   // Method PrintStream.println:(LString;)V
```

Constants in Java

Constants for the compiler

- Literal values (primitive + String)
- static final primitive + String
- Computations involving only constants
  - No method call

Static block ?

Static final initialized in the static block are not constant for the compiler

```
static final int MAGIC;
static {
  MAGIC = 42;
}
```

```
0: getstatic #7          // Field System.out:LPrintStream;
3: getstatic     #13   // Field MAGIC:I
6: invokevirtual #19   // Method java/io/PrintStream.println:(I)V
```

Are they constant at runtime (for the JIT) ?

Perf – static init primitives

Static final initialized in the static block are constant at runtime

```
Benchmark                              Mode  Cnt  Score   Error  Units
ConstantInStaticInitBench.magic        avgt    5  0,309 ± 0,010  ns/op
ConstantInStaticInitBench.magic_block  avgt    5  0,311 ± 0,005  ns/op
ConstantInStaticInitBench.string       avgt    5  0,309 ± 0,004  ns/op
ConstantInStaticInitBench.string_block avgt    5  0,310 ± 0,004  ns/op
```

Accessing a constant object field

Is a field of a constant object a constant ?

```
class Person { final String name; … }

// or record Person(String name) { }

static final Person PERSON = new Person("John");

void main() {
  System.out.println(PERSON.name);  // constant ?
}
```

Perf – static init objects

Fields of a constant record are constant

```
Benchmark                                      Mode  Cnt  Score   Error  Units
ConstantInStaticInitObjectBench.person         avgt    5  0,423 ± 0,044  ns/op
ConstantInStaticInitObjectBench.person_record  avgt    5  0,307 ± 0,005  ns/op
```

JEP draft: Prepare to Make Final Mean Final

Author	Ron Pressler & Alex Buckley
Owner	Ron Pressler
Type	Feature
Scope	SE
Status	Submitted
Component	core-libs
Discussion	jdk dash dev at openjdk dot org
Reviewed by	Alan Bateman, Brian Goetz
Created	2025/02/06 10:25
Updated	2025/04/26 07:43
Issue	8349536

Summary

Issue warnings about uses of *deep reflection* to mutate `final` fields. The warnings aim to prepare developers for a future release that ensures integrity by default by restricting `final` field mutation; this makes Java programs safer and potentially faster. Application developers can avoid both current warnings and future restrictions by selectively enabling the ability to mutate `final` fields where essential.

Accessing a constant List element

Is an element of a constant list a constant ?

```
static final List<String> LIST;

static {
  var list = new ArrayList<String>();
  list.add("Am i a constant ?");
  LIST = list;

  // or LIST = List.of("Am i a constant ?");
}

void main() {
  System.out.println(LIST.getFirst());  // constant ?
}
```

Perf – static init list

Elements of a constant List.of() are constant

```
Benchmark                                    Mode  Cnt  Score   Error  Units
ConstantInStaticInitListBench.arrayList      avgt    5  0,726 ± 0,006  ns/op
ConstantInStaticInitListBench.list_of        avgt    5  0,309 ± 0,009  ns/op
```

Lazy init of constants in Java

In Java, classes are loaded lazily

- So lazy init of constants by default !

If initialization is slow or there are a lot of fields

Refactor as an afterthought

- A library API is already published
- An application uses a facade

static final Stable Value

DEMO !

(2)

static Stable Value Supplier

Lazy constant initialization

```
private static final Supplier<Database> SUPPLIER =
    StableValue.supplier(() - >  new Database(… ));

public static Database getDatabase() {
  return SUPPLIER.get();
}
```

```
Supplier<T> StableValue.supplier(Supplier<T>)
```

Stable Value List

StableValue.list(size, IntFunction<E>)

- A list of lazy initialized elements
- Elements are modifiable (List.set)
- List is not structurally modifiable (List.add/remove)

Stable Value Map

StableValue.map(Set<K> keys, Function<K,V>)

- A map of lazy initialized values
- Values are modifiable (Map.put/replace)
- Map is not structurally modifiable (Map.put/remove)

Perf – stable value

Lazy list and map values are/should constant

```
Benchmark                                              Mode  Cnt  Score   Error  Units
ConstantStableValueBench.stable_supplier_string  avgt    5  0,312 ± 0,002  ns/op
ConstantStableValueBench.stable_value_list       avgt    5  0,313 ± 0,001  ns/op
ConstantStableValueBench.stable_value_map        avgt    5  0,313 ± 0,001  ns/op
ConstantStableValueBench.stable_value_string     avgt    5  0,313 ± 0,001  ns/op
ConstantStableValueBench.string                  avgt    5  0,313 ± 0,001  ns/op
```

TLDR;

Lazy initialization in Java

Java 25 provides a new API StableValue (preview)

- Faster than using a synchronized
- Easier to use than the Double Check Locking
- Should behave as final after initialization (Not Yet !)
- Behave as a constant if static final

https://github.com/forax/stable-value

Future ??

Leyden

StableValue can be precomputed and stored inside the AOT Cache ??

Panama jextract (bridge Java <-> C)

Currently uses index instead of VarHandle because initializing a VH is slow / uses memory

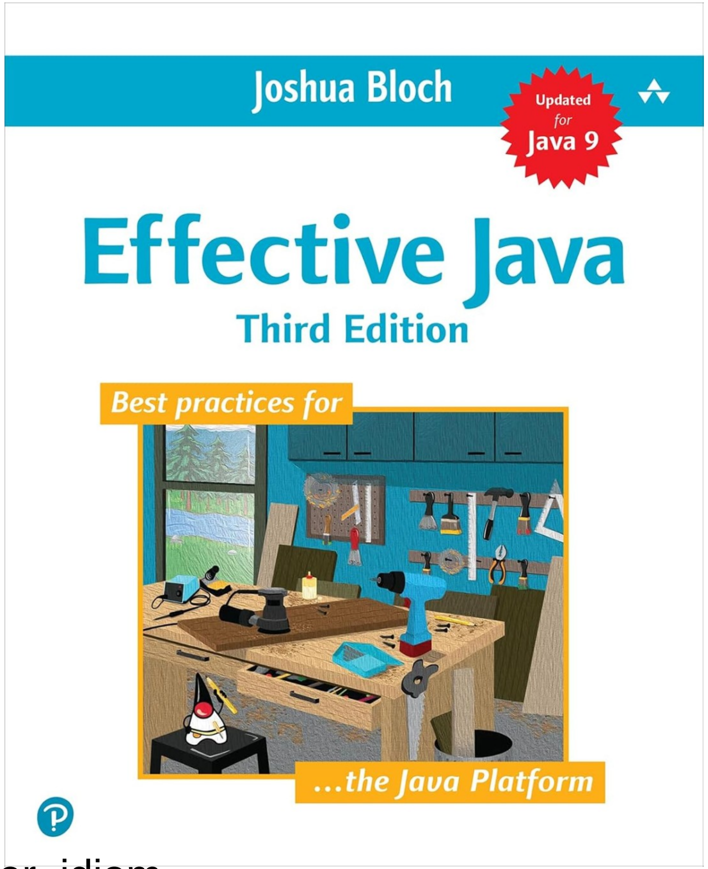
Any questions ?

Supplementary slides

Effective Java

Use lazy class init of Java to initialize a value lazily

=> Initialization on demand idiom



https://en.wikipedia.org/wiki/Initialization-on-demand_holder_idiom

Class holder idiom

Initialization-on-demand holder idiom

```
public static Database getDatabase() {
  public enum Holder {
    ;
    static final Database DB =
      new Database(...);
  }
  return Holder.DB;
}
```

This is the Java 15+ version, with a local enum !

Perf – class holder idiom

Value of the class holder idiom is a constant

```
// Benchmark                                        Mode  Cnt  Score   Error  Units
// LazyStaticInitBench.lazy_class_string            avgt    5  0,312 ± 0,005  ns/op
// LazyStaticInitBench.lazy_dcl_string              avgt    5  0,726 ± 0,009  ns/op
// LazyStaticInitBench.lazy_synchronized_string     avgt    5  5,311 ± 0,042  ns/op
// LazyStaticInitBench.string                       avgt    5  0,313 ± 0,002  ns/op
```

Stable Value

@Stable is not safe

Changing the value multiple times is not allowed

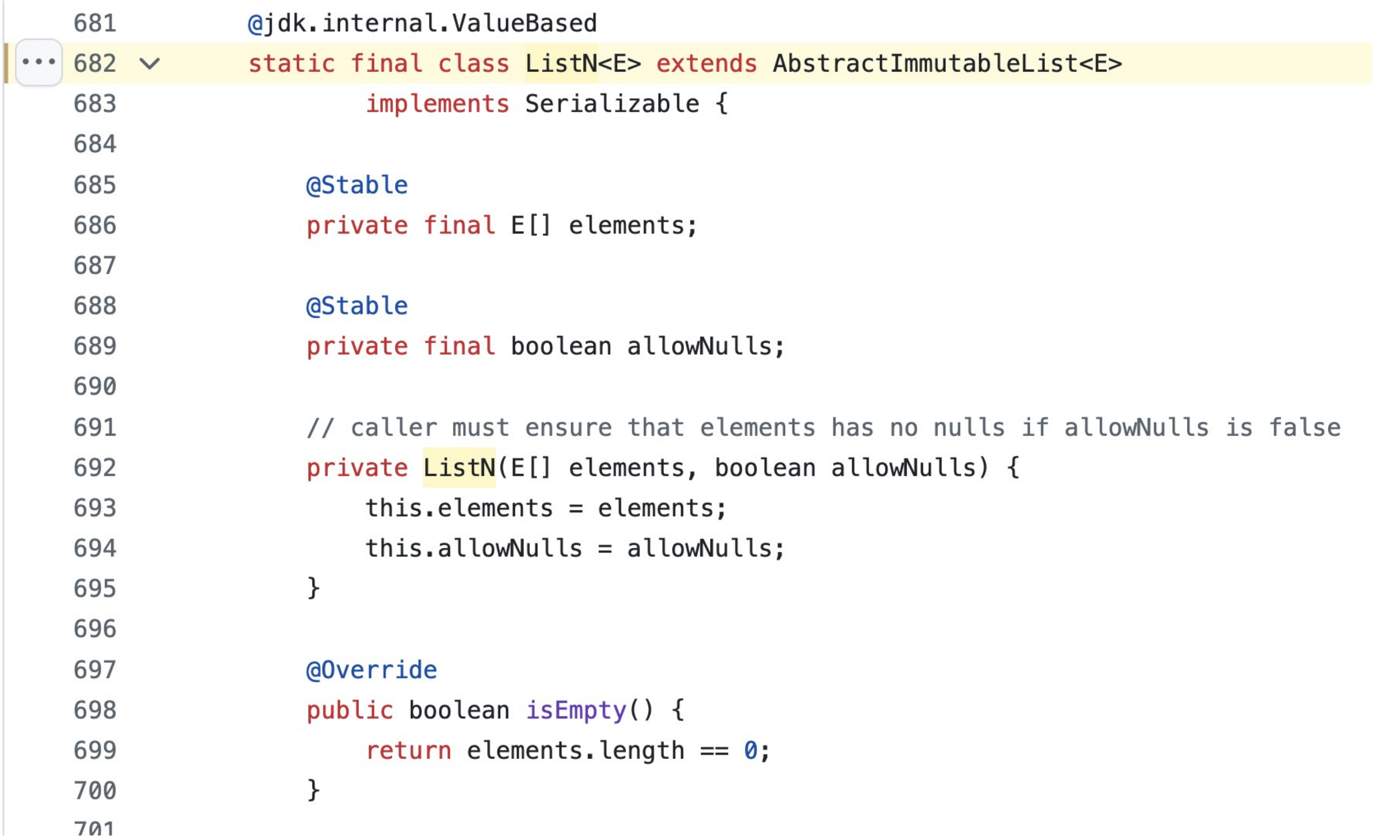
- But not enforced

Public API that provides safe lazy initialization patterns using @Stable

More efficient in resources than the class holder idiom

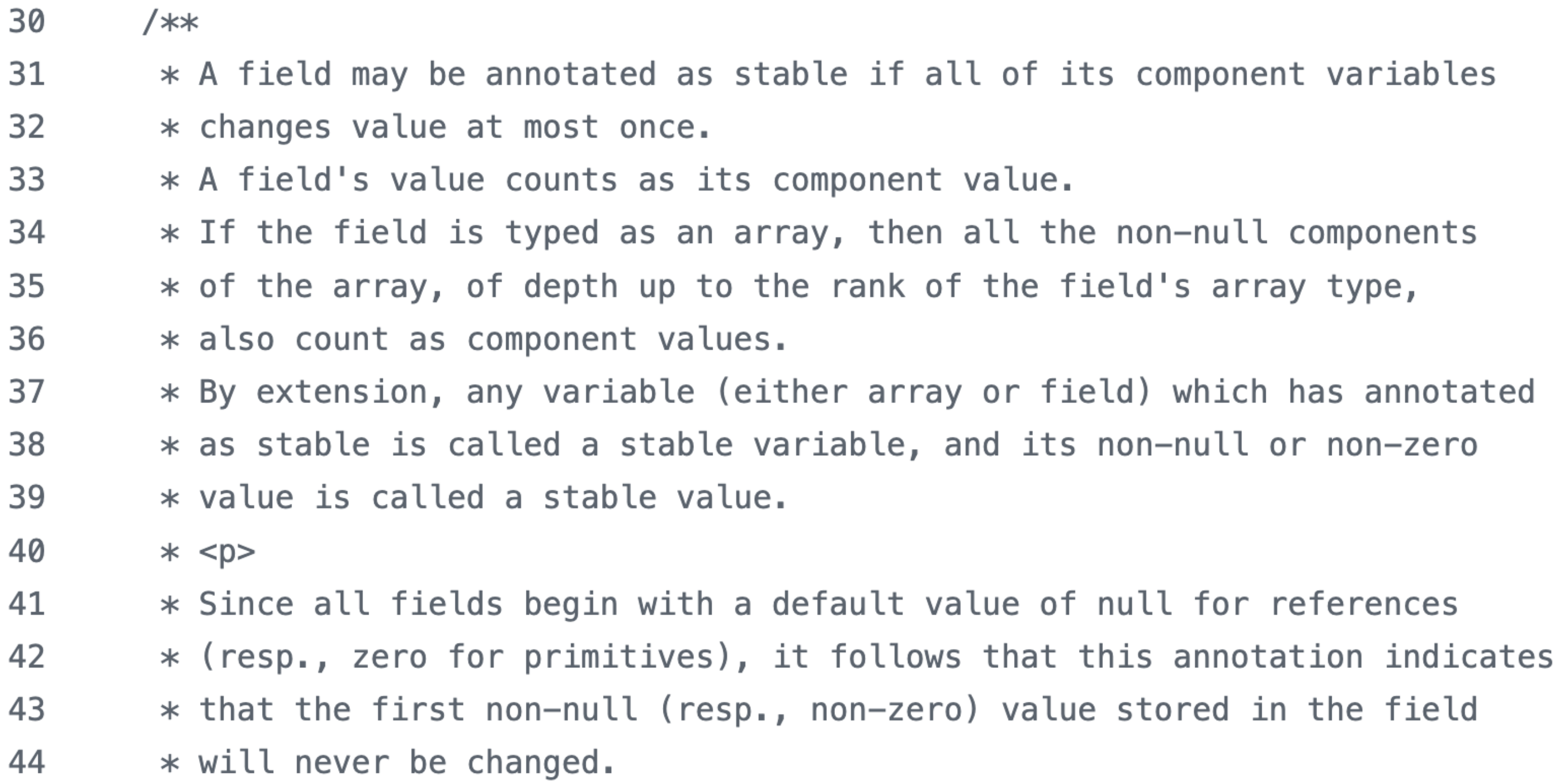
How List.of() works ?

Use an internal annotation @Stable



```
    @jdk.internal.ValueBased
    static final class ListN<E> extends AbstractImmutableList<E>
            implements Serializable {

        @Stable
        private final E[] elements;

        @Stable
        private final boolean allowNulls;

        // caller must ensure that elements has no nulls if allowNulls is false
        private ListN(E[] elements, boolean allowNulls) {
            this.elements = elements;
            this.allowNulls = allowNulls;
        }

        @Override
        public boolean isEmpty() {
            return elements.length == 0;
        }

```

@Stable



```
    /**
     * A field may be annotated as stable if all of its component variables
     * changes value at most once.
     * A field's value counts as its component value.
     * If the field is typed as an array, then all the non-null components
     * of the array, of depth up to the rank of the field's array type,
     * also count as component values.
     * By extension, any variable (either array or field) which has annotated
     * as stable is called a stable variable, and its non-null or non-zero
     * value is called a stable value.
     * <p>
     * Since all fields begin with a default value of null for references
     * (resp., zero for primitives), it follows that this annotation indicates
     * that the first non-null (resp., non-zero) value stored in the field
     * will never be changed.
```